

BEST AVAILABLE COPY

Docket No. 205,413

41528



IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Application of : **TSARFATI**

Serial No.: 10/005,030

:

Group Art Unit: 2124

:

Filed : December 3, 2001 : Examiner: Satish S. Rampuria

:

For : GENERIC FRAMEWORK FOR EMBEDDED SOFTWARE
DEVELOPMENT

Honorable Commissioner for Patents

P.O. Box 1450

Alexandria, Virginia 22313-1450

DECLARATION UNDER 37 CFR 1.131

Sir:

I, the undersigned, Yoram Tsarfati, hereby declare as follows:

1) I am the Applicant in the patent application identified above, and am the inventor of the subject matter described and claimed in claims 1-48 therein.

2) Prior to May 18, 2001, I reduced my invention to practice, as described and claimed in the subject application, in Israel, a WTO country. I implemented the invention in the form of software code, which runs on an embedded CPU in a generic communication line card produced by my employer, Corrigent Systems. The code comprised a number of generic application handler modules, which were designed to be used by

US 10/005,030

Declaration under 37 C.F.R 1.131 by Tsarfati

programmers in creating proprietary handlers for new line cards as needed.

3) As evidence of the reduction to practice of the present invention, I attach hereto a number of programmer guides, which were prepared by software programmers in our company to describe our generic application handler modules and instruct other programmers in creating proprietary handlers using these modules. The following documents are attached:

- Exhibit A: Generic Performance Monitoring Handler.
- Exhibit B: Generic Configuration Handler.
- Exhibit C: Generic Maintenance Handler.
- Exhibit D: Generic Alarm Handler.
- Exhibit E: A report generated by ClearCase, a tool we use to manage and organize software versions, showing logging of the documents presented above in Exhibits A-D. The dates that are blacked out in this Exhibit are prior to May 18, 2001.

4) The following table shows the correspondence between the elements of the method claims in the present patent application and elements of the material in the exhibits:

Claim 1	Exhibits A-D
1. A method for producing embedded software, comprising:	Exhibit A-D contain instructions on programming a processor that is embedded in a generic line card.

Declaration under 37 C.F.R 1.131 by Tsarfati

providing one or more generic application handler programs, each such program comprising computer program code for performing generic application functions common to multiple types of hardware modules used in a communication system;	Section 3.2 on page 5 of Exhibit A describes the performance monitoring (PM) generic part. Similar generic parts are described in Exhibits B-D. Monitoring, configuration and maintenance functions are common to nearly all types of hardware modules that are used in communication systems.
generating specific application handler code to associate the generic application functions with specific functions of a device driver for at least one of the types of the hardware modules; and	Section 3.3 on page 5 of Exhibit A describes the API that is provided as a base for the programmer to develop a proprietary handler. The architecture, including the device driver, is shown in the figure on this page. Section 4.3.2 on pages 9-10 gives details of the API.
compiling the generic application handler programs together with the specific application handler code to produce machine-readable code to be executed by an embedded processor in the at least one of the types of the hardware modules.	Section 4.4. on pages 10-11 of Exhibit A lists source and header files. It is inherently clear that in order for the program to run on the processor, these files must be compiled.

US 10/005,030

Declaration under 37 C.F.R 1.131 by Tsarfati

Claim 2	
2. A method according to claim 1, wherein providing the generic application handler programs comprises providing an application program interface (API) to enable a system management program in the communication system to invoke the generic application functions.	Section 2 on page 4 of Exhibit A states that the PM handler provides an API to other handlers (which may include the system management program).
Claim 3	
3. A method according to claim 2, wherein the one or more generic application handler programs comprise a plurality of generic application programs, and wherein providing the API comprises enabling one of the generic application programs to invoke the generic application functions of another of the generic application programs.	See section 2, page 4, of Exhibit A, as noted in reference to claim 2.

Claim 4	
4. A method according to claim 1, wherein providing the generic application handler programs comprises providing a performance monitoring handler, including a performance monitoring function for counting selected events relating to performance of the hardware modules.	Exhibit A describes a performance monitoring handler. Event counters are listed in the first table in section 4.2.1.
Claim 5	
5. A method according to claim 4, wherein generating the specific application handler code comprises specifying a register in one of the types of the hardware modules whose contents are to be passed to the performance monitoring function for counting.	Section 4.3 on page 8 of Exhibit A describes an API to other handlers, which includes counter and collection functions for use in collecting register contents from other modules.

US 10/005,030

Declaration under 37 C.F.R 1.131 by Tsarfati

Claim 6	
6. A method according to claim 4, wherein providing the generic application handler programs further comprises providing an alarm handler, and wherein providing the performance monitoring handler comprises providing a programmable performance threshold and an alarm invocation function, such that when a count of the selected events exceeds the threshold, the performance monitoring handler sends an alarm message to the alarm handler.	The first table on page 7 (section 4.2.1) of Exhibit A shows alarm counters and thresholds.
Claim 7	
7. A method according to claim 1, wherein providing the generic application handler programs comprises providing a maintenance handler, including a testing function for detecting failures in the hardware modules.	Exhibit C describes the maintenance handler. Testing functions are described in section 4.1 on page 7.

US 10/005,030

Declaration under 37 C.F.R 1.131 by Tsarfati

Claim 8	
8. A method according to claim 7, wherein generating the specific application handler code comprises associating the testing function with at least one of a self test and a sanity test of a component in the at least one of the types of the hardware modules.	Self test and sanity test functions are listed in section 4.2.1 on page 7 of Exhibit C.
Claim 9	
9. A method according to claim 7, wherein providing the generic application handler programs further comprises providing an alarm handler, and wherein providing the maintenance handler comprises providing an alarm invocation function, such that when one of the failures is detected, the maintenance handler sends an alarm message to the alarm handler.	Alarm triggering is described in the second paragraph of section 4.1 on page 7 of Exhibit C.

US 10/005,030

Declaration under 37 C.F.R 1.131 by Tsarfati

Claim 10	
10. A method according to claim 1, wherein providing the generic application handler programs comprises providing a configuration handler, for holding configuration and state information regarding components of the hardware modules.	Exhibit B describes the configuration handler. Functions of this handler are described in section 2 on page 5.
Claim 11	
11. A method according to claim 1, wherein providing the generic application handler programs comprises providing an alarm handler, including functions for receiving and responding to alarm messages generated by others of the application handler programs.	Exhibit D describes the alarm handler. The flow of alarm messages from other handlers is shown in the figure on page 8.

US 10/005,030

Declaration under 37 C.F.R 1.131 by Tsarfati

Claim 12	
12. A method according to claim 11, wherein providing the alarm handler comprises providing a programmable prioritization function, for determining an order of priority for processing of the alarm messages by the alarm handler.	A table for setting alarm priorities is shown at the top of page 12 in section 4.4 of Exhibit D.
Claim 13	
13. A method according to claim 11, wherein generating the specific application handler code comprises specifying a component in one of the types of the hardware modules to actuate so as to notify a user of the system of the alarm message.	Actuation of different system functions in response to alarms is described in the third paragraph of section 4.4, on page 11 of Exhibit D.

US 10/005,030

Declaration under 37 C.F.R 1.131 by Tsarfati

Claim 14	
14. A method according to claim 11, wherein generating the specific application handler code further comprises specifying one of the generic application functions of another of the generic application handler programs to invoke in response to the alarm message.	Section 4.3.1 on page 9 of Exhibit D describes an API between the alarm handler and other handlers, which may be used to invoke responses to alarms by the other handlers.
Claim 15	
15. A method according to claim 11, wherein responding to the alarm messages comprises sending a notification of the alarm message from the alarm handler to a system management program.	Section 2 on page 5 of Exhibit D states that the alarm handler notifies the NMS (a system management program) by sending a trap when an alarm occurs.

US 10/005,030

Declaration under 37 C.F.R 1.131 by Tsarfati

Claim 16	
16. A method according to claim 1, wherein generating the specific application handler code comprises defining specific elements to be handled by the generic application functions for the at least one of the types of the hardware modules, and registering one of the specific functions of the device driver for use in handing each of the defined specific elements.	Section 4.3.2 on page 9 of Exhibit A describes an API that may be used to designate the specific actions to be performed by the proprietary part of the handler.

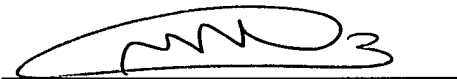
5) Claims 17-48 recite apparatus and a computer software product, with limitations similar to those of method claims 1-16. Based on the similarity of subject matter between the method, apparatus and software claims, it can similarly be demonstrated that we conceived the entire invention recited in claims 17-48 prior to May 18, 2001.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and conjecture are thought to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the

US 10/005,030

Declaration under 37 C.F.R 1.131 by Tsarfati

United States Code and that such willful false statements may jeopardize the validity of the application of any patent issued thereon.

A handwritten signature in black ink, consisting of a series of loops and a final '3' at the end, positioned above a horizontal line.

Yoram Tsarfati, Citizen of
Israel

79 Beit Hanania

Hof Hacarmel 37807

Israel

Date:

12/7/04

EXHIBIT A



Generic Performance Monitoring Handler
Programmer Guide
For generic Line Card

REVISION:

AUTHOR:	<u>Raquel Barzilay</u>	DATE		SIGNATURE	_____
SYSTEM:	_____	DATE	_____	SIGNATURE	_____
APPROVED:	_____	DATE	_____	SIGNATURE	_____

FILE : PM Handler PDR

DISTRIBUTION LIST:

CORRIGENT CONFIDENTIAL AND PROPRIETARY DATA



History Change:

Rev.	Date	Change Description	Author



Table of contents

1. Document Overview	4
2. Module Overview	4
3. Module Architecture	4
3.1 Data Base and DB API.....	4
3.2 PM Generic Mechanism and PM API.....	5
3.3 Proprietary API.	5
4. Module Implementation	6
4.1. Control flow	6
4.2. DB & Data structures.....	6
4.2.1. Data structures.....	6
4.2.2. Databases.....	7
4.3. API	8
4.3.1. API to other handlers	8
4.3.2. API between the generic and the proprietary parts	9
4.4. Detailed implementation	10
4.4.1. Source Files	10
4.4.2. Header Files.....	11
5. How to start	11
5.1. Define the Interval Data.....	11
5.2. Implement the Update function.....	11
6. Module Integration	11



1. Document Overview

This document mainly contain two parts. One to describe the architecture and implementation of the generic Performance Monitoring handler. The second, a manual how to use the generic handler to build a proprietary Performance Monitoring handler.

For a programmer that is itching to start his proprietary handler is recommended to read section 2 & 3 before jumping to section 5. Section 3 describe in more details the implementation of the generic handler.

2. Module Overview

This Handler provides a mechanism to maintain a user defined Performance Monitoring data base, update it periodically and allow data retrieval from it. It also provides an API to other handlers. The generic mechanism uses application functions (provided by the programmer of the proprietary handler) which allow data retrieval from the HW and data base update.

The PM mechanism periodically updates the data base by reading data from the HW. The decision whether an interval is finished can either be done independently (after counting a configurable number of seconds) or event driven (getting a "close interval" command from another task for example the NMS). The working mode should be configured at the beginning and should not be changed during the handler's work.

The PM API allows other handlers to send different commands to the PM such as Reset, Start, Close Current Interval, Get/Set collection mode, change the collection parameters and more.

A threshold mechanism is also provided. For each PM counter there is a set of thresholds configured by the user. When each threshold is set, an alarm code must also be to be sent in case the counter reaches the corresponding threshold.

3. Module Architecture

The module is comprised of the follwing components:

3.1 Data Base and DB API

The DB creation and handling is done by this sub-module. It provides an API which can only be accessed by the PM Generic Mechanism. It allows the creation and handling of the DB in which the PM will be saved. A node in the DB contains the interval number, the actual data (which is only "understood" by the proprietary part) and a pointer to the next node.



3.2 PM Generic Mechanism and PM API

This is the engine of the whole PM mechanism. It is comprised of a PM task, a PM message exchange and a PM timer. The PM timer runs periodically. When it expires it sends a message to the PM task with a PM_TIMER_EXPIRED event. The task waits forever for such a message and when it arrives, it performs an update operation.

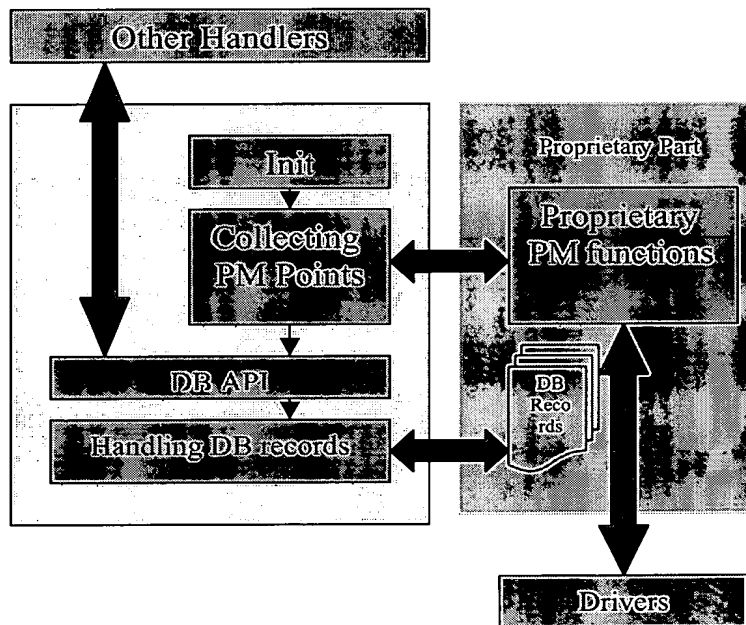
This sub-module also includes an API which is exported to all the other handlers in order to retrieve PM data and perform actions such as reset or start the PM mechanism.

This sub-module provides the data structure to be saved in the data base (LC dependent) and the corresponding threshold structure.

3.3 Proprietary API.

This API is provided as a base for the programmer of the proprietary handler to complete its implementation. It includes all the Line Card dependent functions and configuration parameters. These parameters are provided with the most appropriate defaults but must be adjusted by the programmer of the proprietary handler.

This API can be used by the PM Generic Mechanism and by the Data Base.





4. Module Implementation

4.1. Control flow

The mechanism starts by the call of an initialization function PM_Init(). This function creates the PM task, the PM message exchange and the PM timer and starts the timer.

It also calls DB_Create() API in order to create the PM DB and application init function applPMInit() in order to perform the necessary initializations of the proprietary handler.

Once the PM task, message queue and timer are started, the periodic update mechanism is running. Every second (configurable in seconds) the timer sends a PM_TIMER_EXPIRED message to the task. The task receives this message and calls the generic update function. This function calls the application update function which reads data from the different HW components and updates it into the DB. The access to the DB is performed through the PM Generic Handler.

If the mechanism is configured to work independently then after 900 seconds (the number is configurable in seconds) the current PM interval is closed. If it is configured to close the interval when an event is sent, then only when requested the interval will be closed. The request is done by setting the closeInterval flag which is accessed via an API.

When an update is done each counter is checked to see if one of its thresholds has been reached. In case it has, an alarm message is sent to the alarm handler.

4.2. DB & Data structures

4.2.1. Data structures

Counter types: EPM_COUNTERS(appl)		
Variable Name	Variable Type	Description
EV_PM_COUNTER0	Enum type	Counter type 0 (example)
EV_PM_COUNTER1	Enum type	Counter type 1 (example)
EV_MAX_NUMBER_OF_COUNTERS	Enum type	Max number of counters in the LC

Interval Data: SPM_COUNTERS(appl)		
Variable Name	Variable Type	Description
TimeInSecs	CS_UINT16	Number of elapsed seconds in this interval
PM_counter[EV_MAX_NUMBER_OF_COUNTERS] [MAX_NUM_OF_INTERFACES]	CS_UINT32	Two-dimensional array of counters: counter type (e.g. CRC) x interface number



Thresholds: S_PM_ALARMS (appl)		
Variable Name	Variable Type	Description
thresh[MAX_NUM_OF_THRESH]	CS_UINT32	A number of thresholds for each counter (default = 5)
alarm[MAX_NUM_OF_THRESH]	CS_UINT8	An alarm to send when each threshold is reached

Thresholds: S_PM_THRESHOLDS (appl)		
Variable Name	Variable Type	Description
PM_thresh[EV_MAX_NUMBER_OF_COUNTERS] [MAX_NUM_OF_INTERFACES]	S_PM_ALARMS	Thresholds for PM_counters for each counter type and interface number

Data Base Node: S_PM_INTERVAL (DB)		
Variable Name	Variable Type	Description
IntervalNo	CS_UINT8	Number of interval (0=Current)
Interval	S_PM_COUNTERS	Interval Data

4.2.2. Databases

There are two data bases, two arrays of pointers to functions of the proprietary handler and a number of configuration parameters:

- 1) The PM DB which is an array of intervals of type S_PM_INTERVAL pointed to by pm_CurrentInterval at the beginning and by pm_LastInterval at the end. There are 96 intervals allocated in the array, but the max number of intervals used is stored in maxNumOfIntervals. The actual number of intervals at a given time is stored in numOfIntervals.
- 2) The PM Thresholds data base PM_Thresh of type S_PM_THRESHOLDS.
- 3) The following configuration parameters:

Configuration Parameters		
Variable Name	Variable Type	Description
IntervalNumOfSecs	CS_UINT16	Number of seconds in one interval (default=900)
PmTimerPeriod	CS_UINT8	Length of period in which to update PM (in seconds) (default=1)
CollectionMode	CS_UINT8	INDEPENDENT or EVENT – whether to close an interval independently (according to intervalNumOfSecs or event driven
CloseInterval	CS_UINT8	Flag to force the closing of the current interval next time the PM is updated
numOfIntervals	CS_UINT8	Number of intervals currently saved in the DB
maxNumOfIntervals	CS_UINT8	Max number of intervals to be saved in the DB (configurable by user, and the maximum allowed is 96)



4.3. API

4.3.1. API to other handlers

Syntax: ***void PM_Init(void)***
Initialization function

Syntax: ***void PM_ResetDB(void)***
Reset the DB - clear the current interval and delete all other intervals

Syntax: ***void *PM_GetInterval(CS_UINT8 intervalNo)***
Get the data of a specific interval (by its number)

Syntax: ***CS_UINT32 PM_GetCounter(CS_UINT8 intervalNo,
CS_UINT8 counterNo,
CS_UINT8 interfaceNo)***
Get a specific counter

Syntax: ***void PM_SetCollectionMode(CS_UINT8 independent_or_event)***
Set the collection mode for updating PM (default = independent)

Syntax: ***CS_UINT8 PM_GetCollectionMode(void)***
Get the current collection mode

Syntax: ***void PM_UpdateDB(void)***
Perform an update of the PM DB (from HW)

Syntax: ***void PM_CloseCurrentInterval(void)***
Set a flag to close the current interval when the (1 sec) period is over

Syntax: ***void PM_Start(void)***
Start the whole statistics mechanism NOW (reset and if working in independent mode start the timer again)

Syntax: ***CS_UINT8 PM_ConfigThresholds(CS_UINT8 counterNo,
CS_UINT8 interfaceNo,
CS_UINT8 threshNo,
CS_UINT32 threshValue,
CS_UINT8 alarmType)***



Configure thresholds for sending an alarm. Return RC_OK or RC_OUT_OF_RANGE

Syntax: ***void PM_PrintInterval(CS_UINT8 intervalNo)***
Print an interval of the PM DB

Syntax: ***void PM_PrintDB(void)***
Print the complete PM DB

Syntax: ***void PM_PrintThresholds(void)***
Print the thresholds of the counters and the alarm to be sent when reached

Syntax: ***void PM_SetNumOfIntervals(CS_UINT8 value)***
Set the number of intervals of the DB

Syntax: ***CS_UINT8 PM_GetNumOfIntervals(void)***
Get the number of intervals of the DB

Syntax: ***void PM_SetIntervalNumOfSecs(CS_UINT16 value)***
Set the time length of each interval (in seconds)

Syntax: ***CS_UINT16 PM_GetIntervalNumOfSecs(void)***
Get the time length of each interval (in seconds)

4.3.2. API between the generic and the proprietary parts

This API allows the generic part to perform actions that are LC dependent therefore only the proprietary part can perform them.

Syntax: ***void initPMFunctions (void)***
Init the collection mode (LC specific).
Also perform some initializations necessary for working.

Syntax: ***void applUpdateFromHW(
S_PM_COUNTERS *IntervalData)***
Collect all the counters from the HW and return a structure containing the new interval data.

Syntax: ***void applClearIntervalData(
S_PM_COUNTERS *IntervalData)***
Clear (set to 0) a given interval



Syntax: ***void applPrintIntervalData(***
S_PM_COUNTERS *pIntervalData)
Print the data of a specific interval (for debugging)
When the generic handler is requested to print an interval, it calls this function to print the data, which is known only by the proprietary handler

Syntax: ***void applPrintThresholds(void)***
Print the thresholds for each counter defined in the proprietary handler

Syntax : ***CS_UINT32 applGetCounter(***
CS_UINT8 counterNo,
S_PM_COUNTERS pIntervalData,
CS_UINT8 interfaceNo)
Get a specific counter value.

Syntax: ***void applSetCounter(***
CS_UINT8 counterNo,
CS_UINT8 intervalNo,
CS_UINT8 interfaceNo,
CS_UINT32 value)
Set a specific counter value.

Syntax: ***CS_UINT8 applSetThresh (***
CS_UINT8 counterNo,
CS_UINT8 interfaceNo,
CS_UINT8 threshNo,
CS_UINT32 threshValue,
CS_UINT8 alarmType)
Set a threshold value and the corresponding alarm type.

4.4. Detailed implementation

4.4.1. Source Files

This handler is implemented in three modules:

1. PM_Handler\gen\src\genPM.cpp

This module contains the implementation of the PM Generic Mechanism (task+timer) and all the API exported to the rest of the handlers. The init function `PM_Init()` provided in this module is called by the Main Handler. The flow is described in 4.1. Control flow paragraph.



2. PM_Handler\gen\src\genPMDB.cpp

This module contains the implementation of the PM Data Base and provides an API which is known only by the generic handler

3. PM_Handler\appl\src\PM.cpp

This module contains the Proprietary API provided to the generic mechanism. All actions which are LC dependent are done through this API which should be implemented by the programmer of the proprietary handler.

4.4.2. Header Files

1. ext_inc\genPM.h

Exports PM Generic Mechanism API to all handlers.

2. PM_Handler\ext_inc\PM.h

API between the generic and the proprietary handlers.

3. PM_Handler\gen\inc\inc_genPM.h

PM Generic Mechanism header file

4. PM_Handler\gen\inc\genPMDB.h

Data Base API (exported only to the PM Generic Mechanism)

5. How to start

In this section it will be described how take the generic handler and turn it into a working full PM Handler.

The steps to do are the following:

5.1. Define the Interval Data

Define E_PM_COUNTERS: counter types according to the specific Line Card .

5.2. Implement the Update function

updatePMData function: given a counterNo (counter type) and the corresponding vector of counters from the DB, get data from HW and update the given vector of counters. Function *ApplUpdateFromHW()* will call all the update functions.

6. Module Integration

This module has interaction with the following handlers:

- Main Handler : calls function *PM_Init()* to start the mechanism
- Alarm Handler: when a threshold is reached by a counter, a message is sent to the alarm handler.
- Configuration Handler: There are a few parameters to configure if needed (otherwise they work with a default) described in 4.2.2. Databases. The



threshold table should also be configured with values for the thresholds of each counter and with the alarm types to send in case a threshold is reached.

EXHIBIT B



Generic Configuration Handler
Programmer Guide
For generic Line Card

REVISION:

AUTHOR:	<u>David Rozenberg</u>	DATE	_____	SIGNATURE	_____
SYSTEM:	_____	DATE	_____	SIGNATURE	_____
APPROVED:	_____	DATE	_____	SIGNATURE	_____

FILE : Generic Configuration handler.doc

DISTRIBUTION LIST: _____

CORRIGENT CONFIDENTIAL AND PROPRIETARY DATA



History Change:

Rev.	Date	Change Description	Author



Table of contents

1. Document Overview	5
2. Module Overview	5
3. Module Architecture	6
4. Module Implementation.....	7
4.1. Control flow	7
4.2. DB & Data structures.....	7
4.2.1. Data structures.....	7
4.2.2. Database	7
4.3. API	8
4.3.1. API to other handlers	8
4.3.2. API between the generic and the proprietary parts	9
4.4. Detailed implementation	11
5. How to start.....	11
5.1. Identifying the configurable components	11
5.2. Defining the fields per element.....	12
5.3. Defining the user proprietary functions	13
5.4. Connecting the data to the generic handler	13
5.5. Building the state machine	14
5.5.1. Defining the states	14
5.5.2. Defining the actions	15
5.5.3. Defining the user proprietary functions	15
5.5.4. Building the transfer matrix	15
5.5.5. Connecting the data to the generic handler	16
5.6. Important issues	17
6. Module Integration	17





1. Document Overview

This document mainly contain two parts. One to describe the architecture and implementation of the generic configuration handler. The second, a manual how to use the generic handler to build a proprietary configuration handler.

For a programmer that is itching to start his proprietary handler is recommended to read section 2 & 3 before jumping to section 5. Section 3 describe in more details the implementation of the generic handler.

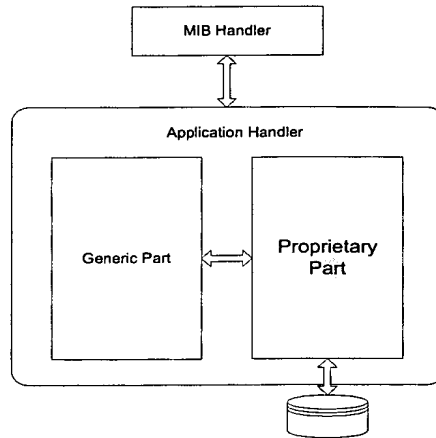
2. Module Overview

This handler provides a known and well defined API for accessing the LC configuration. The API provide a get/set function to any field (only get if it was defined as read-only) in the configuration DB. The handler interpret the request, if it is a get request it return a pointer to the requested field. If it is set request it call the user proprietary set function.

Added to this handler is a generic state machine (it is not part of the configuration, it can be ignored if not needed). The machine can get events, and move to different states according to the event. When entering or exit state, a user proprietary function is called.

3. Module Architecture

The module can be divided into two parts the generic one and the proprietary one.



The generic part will have a well defined API toward other handler, such as the MIB handler. The proprietary part will be written by the user and will do the actual settings to the configuration database. In the proprietary part the user will define all the configurable elements that will be managed and will connect his functions to the generic part (see more details in section 4.3 & 5). From this moment all the calls will be made through the generic handler APIs.

The generic state machine should be defined by states and events. In the proprietary part the user will define all the states and the events of the state machine, and will defined a transfer matrix, from every state what event transfer to what state. The states will be connected into proprietary functions that will be called upon entering or existing state.



4. Module Implementation

4.1. Control flow

The handler have a main initialization function CONF_Init, this function will initialize the internal handler database. After the initialization it require that the user will give for each configurable element a pointer to the actual field and a function to set this field. For the state machine the user will give the transfer matrix.

After this initialization the DB can be accessed by calling the API (see section 4.3 & 5). The state machine will be driven by sending events to it.

4.2. DB & Data structures

The access function to the DB can be divided into get and set function. The get function return a pointer to the actual data, if it was unable to get it, it will return NULL. The set function return RC_OK if the update was succeeded or RC_FAIL otherwise.

The send event function for the state machine return void.

4.2.1. Data structures

S_BOARD, S_INTERFACE, S_COMPONENTS		
Variable name	Variable type	Description
pFunc	void*	A pointer to the setting function.
pppValue	void***	A pointer to two-dimensional array of pointers to the actual data.
numOfElements	CS_UINT8	Number of elements
numOfParamsPer Element	CS_UINT8	Number of fields per element.
isReadOnly	CS_BOOL	The fields are read-only or not

S_STATE		
Variable name	Variable type	Description
stateNum	E_STATES	The state number
pEnterFunction	void*	A pointer to the entering function
pExitFunction	void*	A pointer to the exiting function
moveFunc	E_STATES	An array of transfer, for every event to which state to move.

4.2.2. Database

There are four databases in the system:

- 1) boardDB of S_BOARD records.
- 2) interfaceDB of S_INTERFACE records.
- 3) componentDB of S_COMPONENT records.
- 4) stateMachine of S_STATE records.



The size of the databases depends on how many elements are in each one.

4.3. API

4.3.1. API to other handlers

There are several APIs that enable other handler (especially the MIB handler) to communicate with the handler.

Syntax: *void CONF_Init()*

This is the main initialization function. It should be called once on power-on.

Syntax: *CS_INT8 CONF_SetBoardElement*

(E_BOARD, CS_UINT8, void, CS_UINT8)*

This function get board element, a field, pointer to a data and set the corresponding field to the requested value. The setting is done by calling the user proprietary update function. The function return RC_OK if the update was successfully completed, otherwise RC_FAIL.

Syntax: *CS_INT8 CONF_SetInterfaceElement*

(E_INTERFACE, CS_UINT8, void, CS_UINT8)*

This function get interface element, a field, pointer to a data and set the corresponding field to the requested value. The setting is done by calling the user proprietary update function. The function return RC_OK if the update was successfully completed, otherwise RC_FAIL.

Syntax: *CS_INT8 CONF_SetComponentElement*

(E_COMPONENT, CS_UINT8, void, CS_UINT8)*

This function get component element, a field, pointer to a data and set the corresponding field to the requested value. The setting is done by calling the user proprietary update function. The function return RC_OK if the update was successfully completed, otherwise RC_FAIL.

Syntax: *void* CONF_GetBoardElement*

(E_BOARD, CS_UINT8, CS_UINT8)

This function get board element, a field and return pointer to corresponding field data. The function return NULL if it was unable to find the field.

Syntax: *void* CONF_GetInterfaceElement*

(E_INTERFACE, CS_UINT8, CS_UINT8)



This function get interface element, a field and return pointer to corresponding field data. The function return NULL if it was unable to find the field.

Syntax: *void* CONF_GetComonentElement*
(*E_COMPONENT, CS_UINT8, CS_UINT8*)

This function get component element, a field and return pointer to corresponding field data. The function return NULL if it was unable to find the field.

Syntax: *void CONF_SendAction*
(*E_ACTIONS*)

This function an event and change the state machine state according to it. If the state was actually changed a call is made to the exiting and entering functions of the relevant states.

4.3.2. API between the generic and the proprietary parts

Syntax: *void addBoard*
(*E_BOARD, void*, void**, CS_UINT8, CS_UINT*, CS_BOOL*)

This function get a board element, a pointer to user proprietary set function, a pointer to a two-dimensional array of pointers to the actual data, number of fields in the element, number of elements and does this element is read-only. The function update the board DB so when a set request is done, it will be connected to the set function, and when a get request is done it will be able to retrieve the data.

Syntax: *void addInterface*
(*E_INTERFACE, void*, void**, CS_UINT8, CS_UINT*, CS_BOOL*)

This function get an interface element, a pointer to user proprietary set function, a pointer to a two-dimensional array of pointers to the actual data, number of fields in the element, number of elements, and does this element is read-only. The function update the interface DB so when a set request is done, it will be connected to the set function, and when a get request is done it will be able to retrieve the data.

Syntax: *void addComponent*
(*E_COMPONENT, void*, void**, CS_UINT8, CS_UINT*, CS_BOOL*)



This function get a component element, a pointer to user proprietary set function, a pointer to a two-dimensional array of pointers to the actual data, number of fields in the element, number of elements, and does this element is read-only. The function update the component DB so when a set request is done, it will be connected to the set function, and when a get request is done it will be able to retrieve the data.

Syntax: *void setStateMachine*
(*E_STATES*, *void**, *void**, *E_STATES**, *CS_BOOL*)

This function get a state, a pointer to user proprietary entering function, a pointer to user proprietary exiting function, a pointer to an array of states to move on each event and does this state is the initial state. The function update the state machine DB so when an event is sent, it will know to which state to move and to which function to call.

Syntax: *void initBoardFunction()*

In this function the user should put all his calls to *addBoard*. This function is called by *CONF_Init* function on power-on.

Syntax: *void initInterfaceFunction()*

In this function the user should put all his calls to *addInterface*. This function is called by *CONF_Init* function on power-on.

Syntax: *void initComponentsFunction()*

In this function the user should put all his calls to *addComponent*. This function is called by *CONF_Init* function on power-on.

Syntax: *void fillStateMachine()*

In this function the user should put all his calls to *setStateMachine*. This function is called by *CONF_Init* function on power-on.

Syntax: *void initConfProprietary()*

In this function the user should put all his proprietary initializations. This function is called immediately after all the generic handler initializations are done by *CONF_Init* function on power-on.



4.4. Detailed implementation

The implementation of this handler is quite simple. We have a common header for the generic and the proprietary section toward other handlers, in this header the user define all the configurable components in his system (see detail explanation in section 5). On the generic section we have four databases: boardDB, interfaceDB, componentDB and stateMchine. In the proprietary source file (*conf.cpp*) the user should build his actual DB. We provide a skeleton functions for the user to fill, in those function the user suppose to connect his proprietary DB and functions to the generic API. We provide the user an API to fill the generic handler with a pointers to his proprietary DB and functions (see section 5). We also provide a skeleton function for general purpose initialization that will be called on power-on, if the user need to do some initialization to his data, this is the place. On power-on all those skeleton functions will be called by *CONF_Init*. All other handler may perform calls to the user DB using the well known API without knowing the real implementation of the user in the proprietary section.

5. How to start

In this section it will be described how take the generic handler and turn it into a working full configuration handler. First we should divide the work into several stages: identifying the configurable components, defining the fields per element, defining user proprietary set functions and connecting it to the generic handler.

5.1. Identifying the configurable components

First we should take a look at our system component and see which of them can be configured. Next we should divide them into three categories: interfaces, components and board. The interface elements usually will be all the connectors to the outside world (like a transceiver, framer...). The component elements usually will be elements on the LC (like a FPGA, SERDES ...). The board will contain a logical configuration of the LC (like type, serial number, functionality...). After knowing that, let the handler know about them too. In *genConf.h* file there is a structure *E_BOARD* that hold all the board elements, add them to there (you **must** keep the element *EV_BOARD_LAST* at the end of the structure).

For example let assume we need to configure the CPU, the structure should look like this:

```
typedef enum
{
    EV_BOARD_CPU,
```



```
        EV_BOARD_LAST  
    }E_BOARD;
```

There is a structure *E_INTERFACE* that hold all the interface elements, add them to there (you **must** keep the element *EV_INTERFACE_LAST* at the end of the structure).

For example let assume we need to configure the FRAMER, the structure should look like this:

```
typedef enum  
{  
        EV_INTERFACE_FRAMER,  
        EV_INTERFACE_LAST  
}E_INTERFACE;
```

There is a structure *E_COMPONENT* that hold all the component elements, add them to there (you **must** keep the element *EV_COMPONENT_LAST* at the end of the structure).

For example let assume we need to configure the FPGA, the structure should look like this:

```
typedef enum  
{  
        EV_COMPONENT_FPGA,  
        EV_COMPONENT_LAST  
}E_COMPONENT;
```

5.2. Defining the fields per element

After knowing which elements are in the system, we should define which fields there are in every one of them. The user should build his own DB in the proprietary files (*conf.cpp* & *conf.h*). It is recommended to add in the *genConf.h* enum with all the fields, so any other handler that call the proprietary API will be able to use this enum (Other handler does include the proprietary header but the generic header only, so any defines that will be done there will not be public) but it is not mandatory.

Let assume we need to configure the CPU clock rate, the structure will look something like that:

```
typedef enum  
{  
        EV_BOARD_CPU_CLOCKRATE,  
        EV_BOARD_CPU_LAST  
}E_BOARD_CPU;
```



Let assume we need to configure the FRAMER clock rate and psd, the structure will look something like that:

```
typedef enum
{
    EV_INTERFACE_FRAMER_CLOCKRATE,
    EV_INTERFACE_FRAMER_PSD,
    EV_INTERFACE_FRAMER_LAST
}E_INTERFACE_FRAMER;
```

Let assume we need to configure the FPGA clock rate, the structure will look something like that:

```
typedef enum
{
    EV_COMPONENT_FPGA_CLOCKRATE,
    EV_COMPONENT_FPGA_LAST
}E_COMPONENT_FPGA;
```

In addition the user should build is DB in the proprietary file (*conf.cpp*):

```
static CS_UINT8 cpuClockRate;
static S_FPGA fpga[MAX_FPGA];
static S_FRAMER framer[MAX_FRAMERS];
```

5.3. Defining the user proprietary functions

The actual type and validity check of each field is not known to the generic handler, so a set function should be supplied by the user.

The proprietary set function should be in the following structure:

```
CS_INT8 setFRAMER(CS_INT32* data, CS_UINT8 field, CS_UINT8 element)
```

The data will be a pointer to the value that is requested to set, the field will be which field to set (i.e. *EV_INTERFACE_FRAMER_CLOCKRATE*) and element will be which element (i.e. 0, is the first framer).

The function should return RC_OK if the data was updated successfully otherwise RC_FAIL.

A function should be declared for each element in the system, in our example:

```
setCPU
setFRAMER
setFPGA
```

5.4. Connecting the data to the generic handler

All we have left is to connect the DB and the proprietary function to the generic part. In *conf.cpp* there are skeleton functions for:

```
initBoardFunctions
initInterfaceFunctions
```



initComponentFunctions

In order to connect the generic to the DB, an array of pointers to the actual field should be built. For example:

```
void* framerParamVector[MAX_FRAMERS][EV_INTERFACE_FRAMER_LAST];
CS_UINT8 i;

for (i=0;i<MAX_FRAMER;i++)
{
    framerParamVector[i][EV_INTERFACE_FARMER_CLOCKRATE]=
                                                &framer[i].clockRate;
    framerParamVector[i][EV_INTERFACE_FRAMER_PSD]=&framer[i].psd;
}
```

Now we should call the `addInterface` to add our new interface into the system:

```
addInterface(EV_INTERFACE_FRAMER,(void*)setFRAMER,
(void**)framerParamVector,EV_INTERFACE_FRAMER_LAST,MAX_FRAMER);
```

Those calls are made for each element in board, interface and components categories.

After the calls are made the access to the DB is via the API:

To set framer 1 (remember it is zero base index) clock rate:

```
CS_UINT32 framerClockRate=19;

CONF_SetInterfaceElement(EV_INTERFACE_FRAMER,
                        EV_INTERFACE_FRAMER_CLOCKRATE,&framerClockRate,1);
```

To get framer 0 clock rate;

```
framerClockRate=*(CS_UINT32*)(CONF_GetInterfaceElement(
                        EV_INTERFACE_FRAMER,EV_INTERFACE_FRAMER_CLOCKRATE,0));
```

5.5. Building the state machine

The state machine need several steps:

5.5.1. Defining the states

The first step toward building the states machine is to define the states. In *genConf.h* file there is a structure *E_STATES* that hold all the states, add the required states there (you **must** keep the element *EV_STATE_LAST* at the end of the structure).



For example let assume we need three states shutdown, restart and show time. The structure should look like this:

```
typedef enum
{
    EV_STATE_SHUTDOWN,
    EV_STATE_RESTART,
    EV_STATE_SHOWTIME,
    EV_STATE_LAST
}E_STATES;
```

5.5.2. Defining the actions

In order to move between states, an event should happened. Those events called actions and should be defined in *E_ACTIONS* structure (you **must** keep the element *EV_ACTION_LAST* at the end of the structure).

For example let define the following events:

```
typedef enum
{
    EV_ACTION_SHUTDOWN,
    EV_ACTION_START,
    EV_ACTION_RESTART,
    EV_ACTION_SES_HIGH,
    EV_ACTION_LINK_DOWN,
    EV_ACTION_LINK_UP,
    EV_ACTION_LAST
}E_ACTIONS;
```

5.5.3. Defining the user proprietary functions

Every state can have an entering and exiting function. The user should define the function according to the following prototype:

```
void restartEnter(E_ACTIONS action)
void restartExit(E_ACTIONS action)
```

Where action will be the event that cause the transfer.

5.5.4. Building the transfer matrix

When the states and actions are known, we should define which event in any state transfer to which event. This should be done in the skeleton function *fillStateMachine*.

For example:

```
E_STATES moveMatrix[EV_STATE_LAST][EV_ACTION_LAST];
```



```
//Building the move matrix, could be built in a single const matrix.
moveMatrix[EV_STATE_SHUTDOWN][EV_ACTION_SHUTDOWN]=
EV_STATE_SHUTDOWN;
moveMatrix[EV_STATE_SHUTDOWN][EV_ACTION_START]=
EV_STATE_RESTART;
moveMatrix[EV_STATE_SHUTDOWN][EV_ACTION_RESTART]=
EV_STATE_RESTART;
moveMatrix[EV_STATE_SHUTDOWN][EV_ACTION_SES_HIGH]=
EV_STATE_SHUTDOWN;
moveMatrix[EV_STATE_SHUTDOWN][EV_ACTION_LINK_DOWN]=
EV_STATE_SHUTDOWN;
moveMatrix[EV_STATE_SHUTDOWN][EV_ACTION_LINK_UP]=
EV_STATE_SHOWTIME;

moveMatrix[EV_STATE_RESTART][EV_ACTION_SHUTDOWN]=
EV_STATE_SHUTDOWN;
moveMatrix[EV_STATE_RESTART][EV_ACTION_START]=
EV_STATE_RESTART;
moveMatrix[EV_STATE_RESTART][EV_ACTION_RESTART]=
EV_STATE_RESTART;
moveMatrix[EV_STATE_RESTART][EV_ACTION_SES_HIGH]=
EV_STATE_RESTART;
moveMatrix[EV_STATE_RESTART][EV_ACTION_LINK_DOWN]=
EV_STATE_RESTART;
moveMatrix[EV_STATE_RESTART][EV_ACTION_LINK_UP]=
EV_STATE_SHOWTIME;

moveMatrix[EV_STATE_SHOWTIME][EV_ACTION_SHUTDOWN]=
EV_STATE_SHUTDOWN;
moveMatrix[EV_STATE_SHOWTIME][EV_ACTION_START]=
EV_STATE_SHOWTIME;
moveMatrix[EV_STATE_SHOWTIME][EV_ACTION_RESTART]
=EV_STATE_RESTART;
moveMatrix[EV_STATE_SHOWTIME][EV_ACTION_SES_HIGH]=
EV_STATE_RESTART;
moveMatrix[EV_STATE_SHOWTIME][EV_ACTION_LINK_DOWN]=
EV_STATE_RESTART;
moveMatrix[EV_STATE_SHOWTIME][EV_ACTION_LINK_UP]=
EV_STATE_SHOWTIME;
```

5.5.5. Connecting the data to the generic handler

All the data is ready all we need it to connect it to the generic handler. This is done with the following function:

```
setStateMachine(EV_STATE_RESTART,(void*)restartEnter,
(void*)restartExit,moveMatrix[EV_STATE_RESTART]);
setStateMachine(EV_STATE_SHUTDOWN,(void*)shutdownEnter,
NULL, moveMatrix[EV_STATE_SHUTDOWN],CS_TRUE);
```



In the above case restart state was declared with both enter and exit functions. Shutdown state was declared with only enter function and it is the initial state.

Now on any event all there is left to do is to send it via the API:

```
CONF_SendAction(EV_ACTION_START);
```

5.6. Important issues

For a reminder some important issues:

- 1) The proprietary set function should return RC_OK if the update was successfully completed otherwise RC_FAIL.
- 2) When adding elemetss to the relevant structures, do not delete *EV_BOARD_LAST*, *EV_INTERFACE_LAST*, *EV_COMPONENT_LAST* and leave them at the end of the list.

6. Module Integration

This module is only called by other modules, it does not have any special integration requirements.

EXHIBIT C



Generic Maintenance Handler
Programmer Guide
For generic Line Card

REVISION:

AUTHOR:	<u>David Rozenberg</u>	DATE	_____	SIGNATURE	_____
SYSTEM:	_____	DATE	_____	SIGNATURE	_____
APPROVED:	_____	DATE	_____	SIGNATURE	_____

FILE : Generic Maintenance handler.doc

DISTRIBUTION LIST:

CORRIGENT CONFIDENTIAL AND PROPRIETARY DATA



History Change:

Rev.	Date	Change Description	Author



Table of contents

1. Document Overview	5
2. Module Overview	5
3. Module Architecture	6
4. Module Implementation.....	7
4.1. Control flow	7
4.2. DB & Data structures.....	7
4.2.1. Data structures.....	7
4.2.2. Database	7
4.3. API	8
4.3.1. API to other handlers	8
4.3.2. API between the generic and the proprietary parts	9
4.4. Detailed implementation	10
5. How to start.....	10
5.1. Identifying the HW components	10
5.2. Defining the self-test functions.....	10
5.3. Defining the sanity-test functions	11
5.4. Defining the general functions	12
5.5. Important issues	14
6. Module Integration	14





1. Document Overview

This document mainly contain two parts. One to describe the architecture and implementation of the generic maintenance handler. The second, a manual how to use the generic handler to build a proprietary maintenance handler.

For a programmer that is itching to start his proprietary handler is recommended to read section 2 & 3 before jumping to section 5. Section 3 describe in more details the implementation of the generic handler.

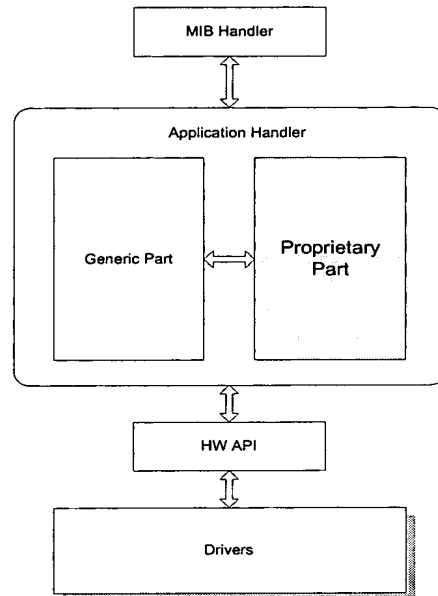
2. Module Overview

This handler provides a mechanism for performing different maintenance requests. Those request can be done on power-up such as system self-test, can come from the NMS such as loopbacks, or periodically such as sanity test. These requests are performed on HW components, the drivers layer provides a HW API package in order to be able to perform those tests. The handler calls the corresponding API and returns the status of the component.



3. Module Architecture

The module can be divided into two parts the generic one and the proprietary one.



The generic part will have a well defined API toward other handler, such as the MIB handler. And a self mechanism to do the generic work (such as periodic sanity test). The proprietary part will be written by the user and will do the actual calls to the HW & HW API. In the proprietary part the user will define all the hardware elements that will be maintained and will connect his functions to the generic part (see more details in section 4.3 & 5). The user will add all the HW component that can be tested, and will supply a function to do those tests. From this moment all the calls will be made through the generic handler APIs.



4. Module Implementation

4.1. Control flow

The handler have a main initialization function MAINT_Init, this function will initialize the internal handler database. After the initialization it require that the user will give for each HW component its test function, for the sanity check the user is require to give the interval between test and an alarm to trigger in case the test fails.

After this initialization the test can be initiated by calling the API (see section 4.3 & 5). The sanity test will be done automatically and will trigger the relevant alarms if necessary.

4.2. DB & Data structures

The return value of the test functions are CS_INT8, a positive or zero value mean that the test was succeed. Negative value mean that the test failed, M_NOTCHECKED=-127 mean that the test did not perform.

4.2.1. Data structures

S_SELFTEST		
Variable name	Variable type	Description
PFunc	Void*	A pointer to the testing function.
LastResult	CS_INT8	The last result

S_SANITYTEST		
Variable name	Variable type	Description
PFunc	Void*	A pointer to the testing function.
Timer	CS_UINT16	The interval between tests (in minutes) zero mean do not test.
Counter	CS_UINT16	Time passed from the last test
Alarm	CS_INT8	The alarm number to trigger if the test fail.
LastResult	CS_INT8	The last result

S_GENERALFUNC		
Variable name	Variable type	Description
PFunc	Void*	A pointer to the testing function.

4.2.2. Database

There are three databases in the system:

- 1) selfTest of S_SELFTEST records.
- 2) sanityTest of S_SANITYTEST records.
- 3) generalFunc of S_GENERALFUNC records.

The size of the databases depends on how many components are in each one.



4.3. API

4.3.1. API to other handlers

There are several APIs that enable other handler (especially the MIB handler) to communicate with the handler.

Syntax: *void MAINT_Init()*

This is the main initialization function. It should be called once on power-on.

Syntax: *CS_INT8 MAINT_SelfTestElement(E_SELFTEST element)*

This function get an HW component and run the test that connected to him. The return value is the return value of the test. Greater or equal to zero is success, negative value is failure.

Syntax: *CS_INT8 MAINT_SelfTestSystem()*

This function run all the tests of the HW components. The return value is the number of element that failed the test. Zero all the components passed the test, other the number of component failed the test.

Syntax: *CS_INT8 MAINT_GetLastSelfTestResult(E_SELFTEST element)*

This function get an HW component and return the last test result of that component. (See *MAINT_SelfTestElement* for return values)

Syntax: *CS_INT8 MAIN_GetLastSanityTestResult(E_SANITYTEST element)*

This function get an HW component that has a periodically sanity test, and return the last test result.

Syntax: *CS_INT8 MAINT_CallFunc(E_GENERALFUNC, void*, void*
void*, void*
void*, void*)*

This function get a general function name and her relevant parameters, it call the connected function and return it return value.



4.3.2. API between the generic and the proprietary parts

Syntax: *void setSelfTestElement(E_SELFTEST, void*)*

This function get and HW component and a pointer to it test function. The function update the self-test DB so when a self-test is requested to that component it will be connected to it test function.

Syntax: *void setSanityTestElement(E_SELFTEST, void*, CS_UINT16, CS_INT8)*

This function get and HW component, a pointer to it sanity test function, an interval for doing the test (the interval is in minutes) and an alarm number to be notify if the test fail. The function update the sanity-test DB so when a sanity-test is requested to that component it will be connected to it test function.

Syntax: *void setGeneralFuncElement(E_SELFTEST, void*)*

This function get a function symbol and a pointer to its function. The function update the generalFunc DB so when a call to a function is requested, it will be connected to it function.

Syntax: *void initSelfTestFunction()*

In this function the user should put all his calls to *setSelfTestElement*. This function is called by *MAINT_Init* function on power-on.

Syntax: *void initSanityTestFunction()*

In this function the user should put all his calls to *setSanityTestElement*. This function is called by *MAINT_Init* function on power-on.

Syntax: *void initGeneralFuncFunction()*

In this function the user should put all his calls to *setGeneralFuncElement*. This function is called by *MAINT_Init* function on power-on.

Syntax: *void initProprietary()*

In this function the user should put all his proprietary initializations. This function is called immediately after all the generic handler initializations are done by *MAINT_Init* function on power-on.



4.4. Detailed implementation

The implementation of this handler is quite simple. We have a common header file between the generic and the proprietary section, in this header the user define all the HW components in his system (see detail explanation in section 5). On the generic section we have three databases: selfTest, sanityTest, generalFunc. In the proprietary source file (*maint.cpp*) we provide a skeleton functions for the user to fill. In those function the user suppose to connect his proprietary functions to the DB. We provide the user an API to fill the DB with a pointers to his proprietary functions (see section 5). We also provide a skeleton function for general purpose initialization that will be called on power-on, if the user need to do some initialization to his data, this is the place. On power-on all those skeleton functions will be called by *MAINT_Init*, this function will also release a task that will wake up every minute to check if there is a sanity check to perform, if there is the proprietary function of that specific sanity check will be called. If the sanity check failed and the alarm is not disabled then an alarm will be send (using the generic alarm handler). All other handler my perform test to HW components or call some general function using the well known API without knowing the real implementation of the user in the proprietary section.

5. How to start

In this section it will be described how take the generic handler and turn it into a working full maintenance handler. First we should divide the work into four stages: identifying the HW components, defining the self-test functions, defining the sanity-test functions and the other general functions.

5.1. Identifying the HW components

First we should take a look at our HW component and see which of them can be tested (for example on power-on) those will insert to the self-test category. Then we need to see which of them need a periodically checking those will insert into the sanity-test category (each component can be in more than one category). Those component who can perform more complex maintenance activity will put into the general function (for example checking the SNR on a link, or doing a loopback).

5.2. Defining the self-test functions

After knowing which component need self-test let the handler know about them too. In *maint.h* file there is a structure *E_SELFTEST* that hold all the components who need to be self-test, add it to there (you **must** keep the element *EV_SELFTEST_LAST* at the end of the structure).



For example let assume we have a transceiver and a FPGA in the system who need self-test. The structure should look like this:

```
typedef enum
{
    EV_SELFTEST_TRANS,
    EV_SELFTEST_FPGA,
    EV_SELFTEST_LAST
}E_SELFTEST;
```

In the file *maint.cpp* you should write the actual function that test the hardware. The return value **must** be zero or greater is the test succeeded, negative value if failed (-127 is reserved).

For the above example let assume you wrote the following functions:

```
testTrans()
testFPGA()
```

Now you should tell the handler about those function. In the *initSelfTestFunctions* you should call *setSelfTestElement* for every HW element. In our example the function should look like this:

```
void initSelfTestFunctions      (void)
{
    setSelfTestElement(EV_SELFTEST_TRANS,(void*)testTRANS);
    setSelfTestElement(EV_SELFTEST_FPGA,(void*)testFPGA);
}
```

That is all! For example you can test your system by calling:
MAINT_SelfTestSystem()

5.3. Defining the sanity-test functions

After knowing which component need periodically sanity-test let the handler know about them too. In *maint.h* file there is a structure *E_SANITYTEST* that hold all the components who need to be periodically tested, add it to there (you **must** keep the element *EV_SANITYTEST_LAST* at the end of the structure).

For example let assume we have a FPGA that needs to be tested every 30 minutes, if it fails it should send the *EV_ALARM_FPGA*. And a PHY that needs to be test every 10 minutes, if it fails it should send the *EV_ALARM_PHY* (The alarms should be declared in the alarm handler). The structure should look like this:

```
typedef enum
```



```
{
    EV_SANITYTEST_FPGA,
    EV_SANITYTEST_PHY,
    EV_SANITYTEST_LAST
}E_SANITYTEST;
```

In the file *maint.cpp* you should write the actual function that test the hardware. The return value **must** be zero or greater is the test succeeded, negative value if failed (-127 is reserved).

For the above example let assume you wrote the following functions:

```
sanityTestFPGA()
sanityTestPHY()
```

Now you should tell the handler about those function. In the *initSanityTestFunctions* you should call *setSanityTestElement* for every HW element. In our example the function should look like this:

```
void initSanityTestFunctions    (void)
{
    setSanityTestElement(EV_SANITYTEST_FPGA,(void*)sanityTestFPGA,
        30,EV_ALARM_FPGA);
    setSanityTestElement(EV_SANITYTEST_PHY,(void*)sanityTestPHY,10,
        EV_ALARM_PHY);
}
```

That is all! The handler will automatically perform the tests in the requested interval, and if it fails it will send the requested alarm. If there is a need to disable a test then you should call it with timeout 0. For example let disable the FPGA sanity test:

```
setSanityTestElement(EV_SANITYTEST_FPGA,(void*)sanityTestFPGA,0,
    EV_ALARM_FPGA);
```

5.4. Defining the general functions

After knowing what component need more complex function let the handler know about them too. In *maint.h* file there is a structure *E_GENERALFUC* that hold all the components who need more complex function, add it to there (you **must** keep the element *EV_GEN_LAST* at the end of the structure).

For example let assume we need to do SNR check for a link, and do LOOPBACK to some link with loopBackType.

The structure should look like this:

```
typedef enum
{
```



```
EV_GEN_LOOPBACK,  
EV_GEN_SNRSCHEK,  
EV_GEN_LAST  
}E_GENERALFUNC;
```

In the file *maint.cpp* you should write the actual function that do requested operation. The function must take 6 pointers as an arguments (if you do not need some of them leave them void*). For the above example let assume you wrote the following functions:

```
CS_INT8 doLoopBack(int* linkNum,int* loopType,void*,void*,void*,void*)  
{  
    printf("Doing loop back to link %d, loop back type %d.\n",  
           *linkNum, *loopType);  
    return 1;  
}  
  
CS_INT8 checkSNR(int* linkNum,void*,void*,void*,void*,void*)  
{  
    printf("SNR on link %d, is 3db.\n",*linkNum);  
    return 3;  
}
```

Now you should tell the handler about those function. In the *initGeneralFuncFunctions* you should call *setGeneralFuncElement* for every function. In our example the function should look like this:

```
void initGeneralFuncFunctions (void)  
{  
    setGeneralFuncElement(EV_GEN_LOOPBACK,(void*)doLoopBack);  
    setGeneralFuncElement(EV_GEN_SNRCHECK,(void*)checkSNR);  
}
```

That is all! For example you can do loopback to link 3 loopType 5:

```
int linkNum=3;  
int loopType=5;  
MAINT_CallFunc(EV_GEN_LOOPBACK, &linkNum, &loopType);
```



5.5. Important issues

For a reminder some important issues:

- 1) The self-test and the sanity-test should return CS_INT8, where zero or greater indicates that the test succeeded, negative value indicates that the test failed, where -127 is a reserved number.
- 2) When adding HW components to the relevant structures, do not delete EV_SELFTEST_LAST, EV_SANITYTEST_LAST, EV_GEN_LAST and leave them at the end of the list.
- 3) The prototype of the general function must contain six pointers, you may not use them all, you may cast them to any other pointer, but you must have six pointers as an arguments.

6. Module Integration

This module integrate only with the alarm handler. In order to test the handler we need to get from the alarm handler all the relevant alarm numbers so we can call them if necessary.

EXHIBIT D



Generic Alarm Handler
Programmer Guide
For generic Line Card

REVISION:

AUTHOR:	<u>Tidhar Tsur</u>	DATE	_____	SIGNATURE	_____
SYSTEM:	_____	DATE	_____	SIGNATURE	_____
APPROVED:	_____	DATE	_____	SIGNATURE	_____

FILE : Generic Alarmhandler.doc

DISTRIBUTION LIST:

CORRIGENT CONFIDENTIAL AND PROPRIETARY DATA



History Change:

Rev.	Date	Change Description	Author



Table of contents

1. Document Overview	5
2. Module Overview	5
3. Module Architecture	6
4. Module Implementation	7
4.1. Control flow	7
4.2. DB & Data structures	8
4.2.1. Data structures	8
4.2.2. Database	9
4.3. API	9
4.3.1. API to other handlers	9
4.3.2. API between the generic and the proprietary parts	10
4.4. Detailed implementation	11
5. How to start	13
6. Module Integration	14





1. Document Overview

This document mainly contains two parts. One to describe the architecture and implementation of the generic alarm handler. The second, a manual how to use the generic handler to build a proprietary alarm handler.

For a programmer that is itching to start his proprietary handler is recommended to read section 2 & 3 before jumping to section 5. Section 3 describes in more details the implementation of the generic handler.

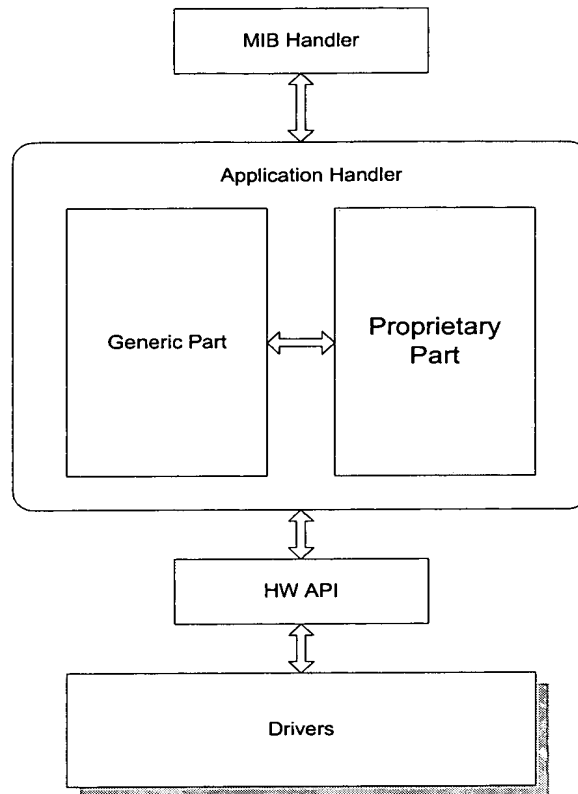
2. Module Overview

This handler provides a mechanism for performing different alarm types the alarm handler is responsible to notify the NMS (usually by trap) of any critical event that occurred in the system. When an event occurs the handler should be activated. The communication is one way, The Alarm handler receives request from all the handlers in the system operate a certain function and sends the relevant trap to the NMS if necessary.

The alarm handler is the only one at the system that responsible for reacting upon the different kind of alarm that flowing at the system including sending the trap.

3. Module Architecture

The module can be divided into two parts the generic one and the proprietary one.

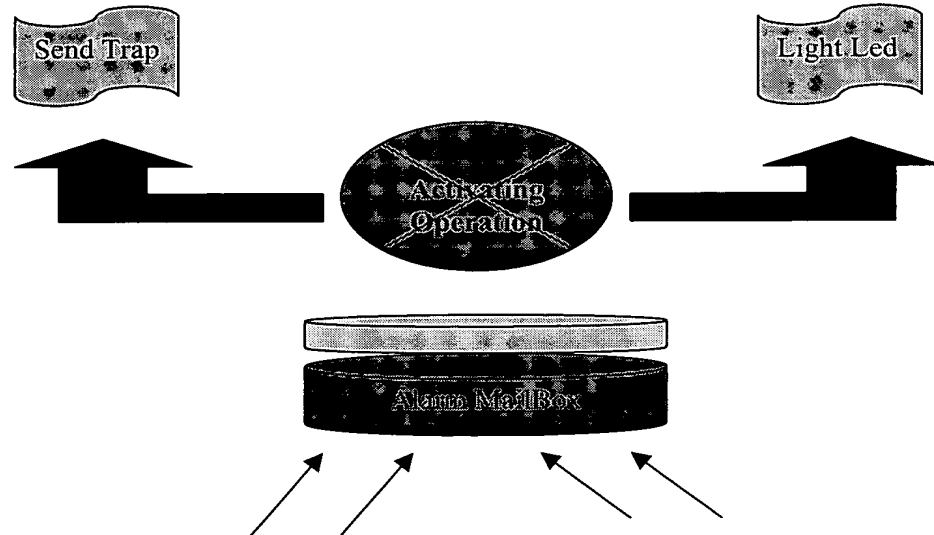


The generic part will have a well-defined API toward other handler, such as the MIB handler. And a self-mechanism to do the generic work (such as waiting on the mailbox for incoming alarm messages). The proprietary part will be written by the user and will do the actual calls to the alarm user handler function. In the proprietary part the user will define for each alarm type its handle function, and will initialize the priorities for each alarm at the system. (See section 4.3 & 5 for more details in).

The user will add all the Alarm types definitions and their relative functions to operate at the case that any system alarm has been activated.

4. Module Implementation

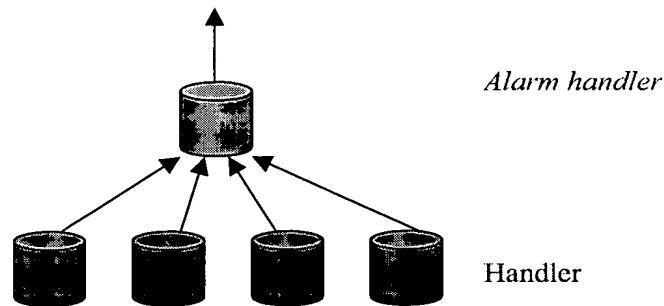
4.1. Control flow



The handlers send a message to the Alarm mailbox, the Alarm handler read the messages one by one at the time it wake up (Or the Alarm handler will wake up upon incoming new alarm message – it can work in those two ways depends on configuration). And send each message to a specific routine. Each of those routines belongs to a specific handler – the handler which generate the alarm, and thus each of those routines operate different actions depends on the handlers sending the Alarm message.

The alarm handler main initialization function - AlarmInit, initialize the internal handler database. After the initialization it creates the alarm handler mailbox. And creates then the alarm task.

Alarm Handler data flow:



4.2. DB & Data structures

The alarm handler maintain four different databases:

- Alarm system status – This struct contain information about the status of the system alarms Active/Not Active, Reprted To NMS/Not Reprted To NMS.
- Alarm system functions - This struct pass on to each alarm the relevant handler function to take care of this alarm.
- Alarm System Enable – A Structure contain the Alarm state – enable/disable by the NMS, The default all the alarms are enable.
- Alarm system Priorities – In that struct the alarm arrange by their Priority.

4.2.1. Data structures

S_ALARM_STATE		
Variable name	Variable type	Description
realAlarmStatus	CS_UINT8	Array contains information on the alarms current atatus.
reportingAlarmStatus	CS_UINT8	Array contains information about reporting of alarms to the NMS.

S_ALARM_FUNCTION_TABLE		
Variable name	Variable type	Description
AlarmFunction	Void*	A pointer to the Alarm handle function.

S_ALARM_ACTIVE		
Variable name	Variable type	Description
alarmEnable	CS_UINT8	Alarms Enable/Disable.



S_ALARM_PRIO_TABLE		
Variable name	Variable type	Description
AlarmPrioListEntry[MaxAlarmType] [MaxPrioEntry]	CS_UINT8	Structer contain the system alarms and the alarms with higher priority at the same entry.

4.2.2. Database

There are four databases in the system:

- 1) alarmState of S_ALARM_STATE records.
- 2) LC_AlarmFunctionTable of S_ALARM_FUNCTION_TABLE records.
- 3) alarmActiveStatus of S_ALARM_ACTIVE records.
- 4) LC_AlarPrioTable of S_ALARM_PRIO_TABLE.

The size of the databases depends on how many components are in each one.

4.3. API

4.3.1. API to other handlers

There are several APIs that enable other handler (especially the MIB handler) to communicate with the Alarm handler.

Syntax: ***void AlarmInit (void)***

This is the main initialization function. It should be called once on power-on.

Syntax: ***int SendMsgToAlarmMBX (CS_UINT8 alarmNum,
CS_UINT32 alarmParam = NULL)***

This function send alarm message to the alarm handler MBX.

Syntax: ***void EnableSpecificAlarm (CS_UINT8 alarmNum)***

The function Enable Specific Alarm. (Upon NMS request.)

Syntax: ***void DisableSpecificAlarm (CS_UINT8 alarmNum)***

The function Disable Specific Alarm. (Upon NMS request.)

Syntax: ***void EnableAllAlarms (void)***

The function Enable All the system Alarms.

Syntax: ***void DisableAllAlarms (void)***

The function Disable All the system Alarms.



4.3.2. API between the generic and the proprietary parts

Syntax: *CS_UINT8 Is_AlarmEnable (CS_UINT8 alarmNum)*

The function Check whether the alarm is active or not.

Syntax: *CS_UINT8 CheckAlarmReportStatus (CS_UINT8 alarmType),*

This function checks whether the alarm has been reported to the NMS.

Syntax: *CS_UINT8 CheckAlarmRealStatus (CS_UINT8 alarmType)*

This function checks whether the alarm has already been activated.

Syntax: *CS_UINT8 Is_alarmTypeOn (CS_UINT8 newAlarmType)*

This function decides whether new incoming alarm is AlarmOff or AlarmOn.

Syntax: *CS_UINT8 SendTrapToNMS (CS_UINT8 newAlarmType)*

This function send trap to the NMS on occasional alarm.

Syntax: *void AlarmPrioTableInitialize (void)*

This function Initialize the Alarm Priority Table.

Each entry of that table represents an Alarm type, and the user with system alarms is filling each entry with higher priority.

Syntax: *AlarmTypeFunctionInitialize(void)*

In this function the user should refer to each alarm type a proprietary handle function. This function is called at the AlarmInit when the entire generic handler is being initialized.



4.4. Detailed implementation

The alarm handler would implement as a TASK with a high priority.

The Alarm handler task will wake up upon a message that has been sent to, through the other handlers.

When the Alarm handler wake up, the first thing it does: pulling up messages from the Alarm MailBox. Then the Alarm handler checked the following:

- Is the Alarm Enable or Disable?
- Is the Alarm message is Alarm Off or Alarm On?

Then it operates the function AlarmWhatToDoNow.

This function are operate the alarm algorithm, there the handler decide what to do with the new incoming alarm message: updating databases, sending a Trap to the NMS through the MIB handler, or operate a specific Led – what ever required handling with the alarm message.

Remark: Traps - the trap mechanism as it implemented in the MIB is change oriented. Its mean that when alarm is on the status field of this alarm is changed and a trap is generated, when this alarm is off the status is changed again and the same trap will be generated again now for clearing this alarm.

The alarm handler will manage all the alarm in the system and will have a logic unit, which will have the ability to decide if to send the traps, or not.

The alarm handler will be able to discard alarms messages neither coming from the handlers because another high priority alarm received and it causes to all the other alarms and there is no meaning to handle them nor sending traps.

In order to implemnt that logic mechanism the Alarm handler maintain the following structure:

```
#define MAX_ALARM_TYPE 10
struct AlarmState{
    unsigned char RealStatus[MAX_ALARM_TYPE];
    unsigned char ReportingStatus[MAX_ALARM_TYPE];
}AlarmStatus;
```

With that structure the Alarm handler can obtain which of the alarms have been activated and have been reported to the NMS and which of them have been activated and havn't been reported to the NMS.

The alarm handler maintains a Priority Alarms DB initilaize by the user.

```
typedef struct {
    CS_UINT8
    AlarmPrioListEntry[MAX_ALARM_TYPE][MAX_PRIORITY_ENTRY];
}S_ALARM_PRIO_TABLE;
```



Each entry in that table represents a specific alarm type and a list of alarms with higher priority in the system.

For instance:

ALARM TYPE	Higher 1	Higher 2	Higher 3	Higher 4	Higher 5
1	0	0	0	0	0
5	1	2	3	4	0

The alarm algorithm Pseudo code:

```
void AlarmMain (void)
{
    int status = CS_TRUE;
    S_ALARM_MESSAGE MsgAlarmBuf,*MsgAlarmBufP;

    MsgAlarmBufP = &MsgAlarmBuf;

    while (status != ERROR)
    {
        // Reading from the Msg Queue till there will be no message inside.
        // status = msgQReceive(AlarmQID, (char *)MsgAlarmBufP,
MAX_ALARM_MSG_LEN, NO_WAIT);
        if (status != ERROR)
        {
            if (Is_AlarmEnable(MsgAlarmBufP->alarmType))
                AlarmWhatToDoNow(MsgAlarmBufP->alarmType);
            else continue;
        }
    }
} // End of AlarmMain

static CS_UINT8 AlarmWhatToDoNow (CS_UINT8 newAlarmType)
{
    CS_UINT8 alarmOffOn;
    CS_UINT8 alarmIndex=1;

    alarmOffOn = Is_alarmTypeOn(newAlarmType);
    if (alarmOffOn)
    {
        if (CheckAlarmRealStatus(newAlarmType))
            return 0;
        alarmState.realAlarmStatus[newAlarmType] = CS_ON;
        while (LC_AlarmPrioTable.AlarmPrioListEntry[newAlarmType][alarmIndex])
        {
            if
(CheckAlarmRealStatus(LC_AlarmPrioTable.AlarmPrioListEntry[newAlarmType][alarmIndex]))
            {
                alarmIndex=1;
            }
        }
    }
}
```



```
        return 0;
    }
    else alarmIndex++;
}
alarmIndex=1;

(* (P_FUNC_CAST)(LC_AlarmFunctionTable.AlarmFunction[newAlarmType]))();
alarmState.reportingAlarmStatus[newAlarmType] = CS_ON;
SendTrapToNMS(newAlarmType);
return 1;
}
else
{
    if (!(CheckAlarmRealStatus(newAlarmType-1)))
        return 0;
    alarmState.realAlarmStatus[newAlarmType-1] = CS_OFF;
    if (CheckAlarmReportStatus(newAlarmType-1))
    {
        alarmState.reportingAlarmStatus[newAlarmType-1] = CS_OFF;
        SendTrapToNMS(newAlarmType);
    }
    return 1;
}
} // End of AlarmWhatToDoNow
```

5. How to start

In this section it will be described how take the generic handler and turn it into a working full alarm handler.

The user should implement the following items:

- Definition of the system alarm type.
- Initialization of the alarms priority table.
- Initializations of the alarms handle function.
- Rewrite the function that send trap to the SSM.

1) The user should continue with list defined at the “AlarmType.h” and define the LC Internal Alarm.

```
For instance:      #define MAX_ALARM_TYPE          4

#define MAINTENANCE_LOPPBACK_FAILED_ON          0x0001
#define MAINTENANCE_LOPPBACK_FAILED_OFF         0x0002
#define PM_UAS_ON                                0x0003
#define PM_UAS_OFF                              0x0004
```



- 2) The User should initialize the alarm priority table using the function: `AlarmPrioTableUserInitialize (void)` inside this function the User should write into the Alarm Priority DB with the function `setAlarmPriorityEntry (CS_INT8 alarmType, S_ALARM_PRIO_ENTRY alarmPrioList)` the user can take the access to insert a whole entry to the DB. The function takes two parameters as an input: `alarmTypeId` and an array of five Bytes in which it defines the alarms, which have a higher priority than the `alarmTypeId`.
For instance the `alarmTypeId = 3`
And the array is `{4,5,0,0,0}`. This means that the `alarmTypeId 3` depends on 4 & 5 and 4,5 are had a higher priority than `alarmTypeId 3`.
- 3) The user should define a function to each alarm type that needed to be handle. In order to obtain a function to specific alarm type, it need to use the function: `AlarmTypeFunctionUserInitialize (void)` there the user can update the Alarm Function DB using the function: `void setAlarmFunction (CS_INT8 alarmType, void* pFunc)` which providing access to the Alarm Function DB. That function gets two parameters as input the `alarmTypeId` and the function that belongs to that alarm.
In the file *AlarmPrio.cpp* the user should write the actual handle functions.
- 4) The user should edit the function `SendTrapToNMS (CS_UINT8 AlarmType)` in accordance to the mechanism of SNMP sending Traps that depends on the agent MIB compiler code.

6. Module Integration

The handlers who engage with the Alarm handler are:

- Performance handler.
- Maintenance handler.
- MIB handler.

In order to send a message to the alarm MBX the system handlers need to use the `ALARM_MN_SendMsgToAlarmMBX` API, which take as input the alarm type and the parameters if necessary.

In order to send a trap to the MIB handler the alarm handler used the function `SendTrapToNMS (CS_UINT8 AlarmType)`

EXHIBIT E – CLEARCASE DOCUMENTATION VERSION CONTROL PRINTOUT

```
M:\cc_vobs\pcc_qa\r&d\RT\line_cards\doc\generic_lc>cleartool lshis -fmt
15.15-%"u %-8.8Nd %n\n"    "pm handler pdr.doc "
RAQUELB [REDACTED] pm handler pdr.doc@@\main\2
RAQUELB [REDACTED] pm handler pdr.doc@@\main\1
RAQUELB [REDACTED] pm handler pdr.doc@@\main\0
RAQUELB [REDACTED] pm handler pdr.doc@@\main
RAQUELB [REDACTED] pm handler pdr.doc@@
```

```
M:\cc_vobs\pcc_qa\r&d\RT\line_cards\doc\generic_lc>cleartool lshis -fmt
15.15-%"u %-8.8Nd %n\n"    "generic configuration handler.doc "
DAVIDR [REDACTED] generic configuration handler.doc@@\main\2
DAVIDR [REDACTED] generic configuration handler.doc@@\main\1
DAVIDR [REDACTED] generic configuration handler.doc@@\main\0
DAVIDR [REDACTED] generic configuration handler.doc@@\main
DAVIDR [REDACTED] generic configuration handler.doc@@
```

```
M:\cc_vobs\pcc_qa\r&d\RT\line_cards\doc\generic_lc>cleartool lshis -fmt
15.15-%"u %-8.8Nd %n\n"    "generic maintenance handler.doc "
RAQUELB [REDACTED] generic maintenance handler.doc@@\main\1
RAQUELB [REDACTED] generic maintenance handler.doc@@\main\0
RAQUELB [REDACTED] generic maintenance handler.doc@@\main
RAQUELB [REDACTED] generic maintenance handler.doc@@
```

```
M:\cc_vobs\pcc_qa\r&d\RT\line_cards\doc\generic_lc>cleartool lshis -fmt
15.15-%"u %-8.8Nd %n\n"    "generic alarm handler.doc "
RAQUELB [REDACTED] generic alarm handler.doc@@\main\1
RAQUELB [REDACTED] generic alarm handler.doc@@\main\0
RAQUELB [REDACTED] generic alarm handler.doc@@\main
RAQUELB [REDACTED] generic alarm handler.doc@@
```

**This Page is Inserted by IFW Indexing and Scanning
Operations and is not part of the Official Record**

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ **BLACK BORDERS**
- ☐ **IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**
- ☐ **FADED TEXT OR DRAWING**
- ☐ **BLURRED OR ILLEGIBLE TEXT OR DRAWING**
- ☐ **SKEWED/SLANTED IMAGES**
- ☐ **COLOR OR BLACK AND WHITE PHOTOGRAPHS**
- ☐ **GRAY SCALE DOCUMENTS**
- ☐ **LINES OR MARKS ON ORIGINAL DOCUMENT**
- ☐ **REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**
- ☐ **OTHER:** _____

IMAGES ARE BEST AVAILABLE COPY.

As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.